Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android

Alexandre Bartel, Jacques Klein, Martin Monperrus, Yves Le Traon

Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android

Alexandre Bartel, Jacques Klein, Yves Le Traon
University of Luxembourg, SnT
Luxembourg, Luxembourg
firstName.lastName@uni.lu

Martin Monperrus
University of Lille
INRIA
Lille, France
martin.monperrus@univ-lille1.fr

ABSTRACT

A common security architecture, called the permission-based security model (used e.g. in Android and Blackberry), entails intrinsic risks. For instance, applications can be granted more permissions than they actually need, what we call a "permission gap". Malware can leverage the unused permissions for achieving their malicious goals, for instance using code injection. In this paper, we present an approach to detecting permission gaps using static analysis. Our prototype implementation in the context of Android shows that the static analysis must take into account a significant amount of platform-specific knowledge. Using our tool on two datasets of Android applications, we found out that a non negligible part of applications suffers from permission gaps, i.e. does not use all the permissions they declare.

1. INTRODUCTION

Android is one of the most widespread mobile operating system in the world accounting 52% market share [13]. More than 300 000 Android applications available on dozens of application markets can be installed by end users. The other side of the coin is that all kinds of malware are waiting to be installed on thousands of Android devices. For instance, Zeus [17] sends banking information to malicious servers. This motivates researchers and engineers to devise security models, architectures and tools that are able to mitigate the malware harmfulness.

The security architecture of Android, the Google Chrome browser extension system and the Blackberry platform, all use a similar security model called the permission-based security model [1]. A permission-based security model can be loosely defined as a model in which 1) each application is associated with a set of permissions that allows accessing certain resources[1]; 2) permissions are explicitly accepted by users during the installation process and 3) permissions are checked at runtime when resources are requested.

This permission model entails intrinsic risks. For instance, not all users may be able to cleverly reject powerful permissions at installation time. Malwares may also use platform vulnerabilities to circumvent runtime permission checks. Finally, applications can be granted more permissions than they actually need, what we call a "permission gap". Malwares can leverage the unused permissions for achieving their malicious goals and have many ways to do so, for instance using code injection or return-oriented programming [6]. Identifying permission gaps means reducing the risks for an application to be compromised, also known as reducing the application attack surface [20].

Let us make an analogy with a firewall. In a correctly configured firewall only the ports that are used are open. All the other ports are closed. However if the firewall is misconfigured, some unused ports remain open and the attack surface of the infrastructure behind the firewall is larger. For instance, let us assume that an information system internally uses a remote shell service on port 544. If port 544 is open on the firewall, an attacker could perform attacks on the remote shell server located behind the firewall. In the same way, an application that requires too many permissions, i.e. that suffers from a permission gap, may allow an attacker who compromised the application to access more resources than he should have.

Permission gaps appear because the process of declaring application permissions is manual and error-prone: Android framework developers manually document which permissions are required for each system resource, and Android application developers manually declare the permissions they *think* are needed. This paper presents an approach to support those *manual* software engineering activities with an *automated* tool. This approach secures permission-based software in the sense that it reduces the attack risks (not in the sense that the resulting software is unattackable).

Our tool, called COPES (COrrect PErmissions Set), proceeds as follows. First, using static analysis, it extracts from the Android framework bytecode a table that maps every method of the API to a set of permissions the method needs to be executed properly. Second, COPES lists all framework methods used by an application, based on static analysis of the application bytecode. Third, COPES com-

[1]Contrary to the traditional Unix permission system where permissions are at the level of users, not applications.

putes the set of permissions that are required for the application to run, which means that all permissions in this set are may be at least once used in the application, and consequently no permission gap remains. Eventually, COPES computes the permission gap as the difference between the declared permissions and the required permission. By listing the permission checks per framework method, COPES can also help Android designers to comprehensively document the framework.

To sum up, the contribution of this paper is an approach to identify and fix permission gaps in permission based software. More specifically:
• We show that the permission-based security model can be expressed within a boolean matrix algebra. This algebra is not specific to Android.
• We present a novel methodology to compute a close approximation of the required permission set and the permission gap based on static analysis, as opposed to concurrent work that uses testing [10].
• We discuss the design and the implementation of the approach for the Android platform.
• We evaluate our approach on 742+679 Android applications and we show that 94+35 applications suffer from a permission gap.

The reminder of this paper is organized as follows. In Section 2 we explain why reducing the attack surface is important and present a short study supporting our intuition. In Section 3 we propose a formalization for permission-based software and a generic method for deriving correct application permission sets. In Section 4 we describe the Android system and its access control mechanisms. Then, in Section 5 we apply the generic method on the Android system. Experiments we conducted and results are presented and discussed in Section 6. We present the related work in Section 7. Finally we conclude the paper and discuss open research challenges in Section 8.

2. THE PERMISSION GAP PROBLEM

This section further details the permission gap problem introduced in Section 1, and presents short empirical facts showing that this problem actually happens in practice.

2.1 Possible Consequence of a Permission Gap

Let us consider an Android application, app_{wrong}, that is able to communicate with external servers since it is granted the INTERNET permission. Moreover, app_{wrong} has declared permission CAMERA while not using it. The CAMERA permission allows the application to take picture without user intervention, i.e. the permission gap consists of one permission: CAMERA. Unfortunately, app_{wrong} uses a native library on which a buffer-overflow exploit has recently been discovered. As a result, through specific payloads, attackers are able to attack devices that are running app_{wrong} in order to take pictures using the device's camera and send them to a remote location on the Internet.

On the contrary, if app_{wrong} did not declare CAMERA, this attack would not have been possible, and the consequences of the buffer-overflow exploit would have been mitigated. As noted by Manadhata [20], reducing the attack surface does not mean no risks, but less risks. In order to show that this example of misconfigured application is not artificial, we now discuss a short empirical study on the declaration of two permissions on 1000+ Android applications.

2.2 Declaration and Usage of Permissions CAMERA and RECORD_AUDIO

We conducted a short empirical study on a 1000+ Android applications downloaded from the Freewarelovers application market[2]. For permissions CAMERA and RECORD_AUDIO, we grepped the source code of the Android framework to approximate the set of methods requiring one of them. These two sets of methods are noted M_{CAMERA} and M_{RECORD_AUDIO}. Then, we computed the list A of all the applications which declare CAMERA or RECORD_AUDIO. Next, we took each application $app \in A$ individually and we checked the application uses at least one method of M_{CAMERA} and M_{RECORD_AUDIO} by analyzing the application's bytecode. If it is not the case, it meant that app is not using the corresponding permission. When this happened, we modified the application manifest that declares the permission and run the application again to make sure that our grepping approximation did not yield false positives.

There are 7/82 applications that declare CAMERA while not using it. Similarly, 3/35 applications declare but do not use RECORD_AUDIO . Those results confirm our intuition: declared permission lists are not always required, and permission gaps indeed exist. Developers would benefit from a tool that automatically infers the set of required permissions and computes the permission gap.

3. ANALYZING PERMISSIONS

In this Section we formalize the concept of permission-based software and propose a generic methodology to compute a mapping from code to permissions that are required for the application to run.

Permission-based software is conceptually divided in three layers: 1) the core platform which is able to access all system resources (e.g. change the network policy), it is generally the operating system; 2) a middleware responsible for providing a clean application programming interface (API) to the OS resources and for checking that applications have the right permissions when they want accessing them; 3) applications built on top of the middleware. They have to explicitly declare the permissions they require. Layers #2 and #3 motivate the generic label "permission-based software". Since the middleware also hides the OS complexity and provides an API, it is sometimes called, as in the case of Android, a "framework".

Let us now discuss those terms more in depth and then show how to infer the list of permissions required by a permission-based application.

3.1 Definitions

Framework A framework $\mathcal{F}$ is a layer that enables applications to access resources available on the platform. We model it as a bi-partite graph between framework API methods and resources.

[2] `http://www.freewarelovers.com/android/`

Example: In the case of Android, $\mathcal{F}$ is the Android 2.2 Java Framework composed of 4071 classes and 126660 methods. To access a resource, an Android application has to make a method call that goes through $\mathcal{F}$.

Permission-based system A permission-based system is composed of at least one framework, a list of permissions and a list of protected resources. Each protected resource is associated with a fixed list of permissions

Entry point An entry point of a framework is a method that an application can use (e.g. public or documented). Constructors are also considered as entry points. We denote $Entry_{\mathcal{F}}$ the set of all entry points of $\mathcal{F}$.
Example: One of the entry points of the Android framework is the method *getAccounts()* from class *AccountManager*.

An application can call any public method of the framework. Some methods accessing some system resources (like an account) are protected by one or more permissions. Let us suppose that the method *getAccounts()* allows access to a set of accounts and is protected by one permission `GET_ACCOUNTS`. An application can successfully call method *getAccounts()* if and only if it declared `GET_ACCOUNTS` in the application-specific list (this list is contained in a "manifest", we shall use this term later in the paper).

Permission A permission is a token that an application needs to access a specific resource. We make no assumptions on permissions, and we consider them as independent (neither grouped, nor hierarchical) .
Example: Developers of an Android application define a list of permissions in a file called the Manifest. To read contact information, the manifest of the application must declare the `READ_CONTACT` permission.

Permissions can be checked at different levels in the system. We call high-level permissions the set $P = \{p_1, p_2, ..., p_n\}$ of permissions that are checked at the framework level. Low-level permissions are permissions that are checked at the operating system level.

High-level permission A high-level permission, is a permission that is only checked at the framework level.
Example: In the case of Android, `READ_CONTACT` is a high-level permission.

Low-level permission A low-level permission is a permission associated with a high-level permission and is checked at a lower level than the framework level.
Example: They are 115 permissions in the Android system, while 8 permissions are checked at a low-level. This shows that the framework is responsible for much of the work related to permissions. Note that if a permission is checked at the operating system level, it is not possible to detect that an application uses it by only analyzing the framework.

Declared permission A declared permission for an application *app* is a permission which is in the permission list of *app*. The set of all declared permission for an application *app* is noted $P_d(app)$.

Required permission A required permission for an application *app* is a permission associated with a resource that *app* uses at least once. The set of all required permissions for an application *app* is noted $P_{req}(app)$.
Example: For an application *app*, if the set of required permissions $P_{req}(app)$ is equal to the set of declared permissions $P_d(app)$, the permission attack surface is minimal.

Inferred permission An inferred permission for an application *app* is a permission that an analysis technique found to be required for *app*. This paper presents such a technique and computes a set of inferred permissions noted $P_{ifrd}(app)$. Depending on the analysis technique used, the inferred permission list may be either an over- or an under- approximation of the required permission list. When using static analysis techniques, the inferred permission list may be an *over approximation* ($P_{req}(app) \subseteq P_{ifrd}(app)$). The inferred permission list may be an *under-approximation* of the required permission list ($P_{ifrd}(app) \subseteq P_{req}(app)$) when using testing techniques (testing-based analysis observes only the executed permissions and potentially misses some permission checks depending on the completeness of the input data).

When developers write manifests, they write $P_d(app)$ by trying to guess $P_{req}(app)$ based on documentation and trial-and-errors. In this paper, we propose to automatically infer a permission list $P_{ifrd}(app)$ in order to avoid this manual and error-prone activity. We take a special care in minimizing the difference between $P_{ifrd}(app)$ and $P_{req}(app)$.

3.2 A Calculus for Permissions

This section describes the permission gap inference as a clean and regular calculus on top of boolean matrix algebra. More importantly, while permission inference is at heart a reachability analysis (does the application reach a permission check?), this calculus factorizes much of the static analysis, hence is much more efficient.

Let *app* be an application. The *access vector* for *app* is a boolean vector AV_{app} representing the entry points of the framework reachable from *app*. Thus, the length of vector AV is the number of entry points of $\mathcal{F}$. An element of the vector is set to *true* if the corresponding entry point is called by the application. Otherwise it is set to *false*. Let us consider a framework with four entry points (e_1, e_2, e_3, e_4), and an application *app* with the following access vector, expressing that *app*'s code may reach e_1, e_2 and e_3 but not e_4:

$$AV_{app} = (1, 1, 1, 0)$$

We define the *permission access matrix* M as a boolean matrix which represents the relation between entry points of the framework and permissions. Rows represent entry points of the framework and columns represent permissions. A cell $M_{i,j}$ is set to *true* if the corresponding entry point (at row i) accesses a resource protected by the permission represented by column j. Otherwise it is set to *false*. For a framework with four entry points (e_1, e_2, e_3 and e_4) and three permissions (p_1, p_2 and p_3), the permission access matrix could

be:

$$M = \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

This means that e_1 and e_2 require permission p_1, e_3 requires no permission and that e_4 requires permission p_2.

Let *app* and $\mathcal{F}$ be an application and a framework respectively. The inferred permissions vector, IP_{app}, is a boolean vector representing the set of inferred permissions for application *app*. We have $IP_{app} = AV_{app} \times M$ by using the boolean operators AND and OR instead of arithmetic multiplication and addition in the matrix calculus. A cell $IP_{app}(k)$ is set to *true* if the permission at index k is required by *app*. Otherwise it is set to *false*. Note that $P_{ifrd}(app)$ is the set of all permissions set to *true* in IP_{app}, i.e. $P_{ifrd}(app)= \{permission_x | IP_{app}(x)\}$. Using AV_{app} and M from the two previous examples, the inferred permissions vector for *app* is:

$$IP_{app} = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$IP_{app} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

Application *app* should declare permissions p_1.

3.3 Extraction of M and AV

In this section we present a methodology to extract the permission access matrix M of a framework $\mathcal{F}$. This methodology is based on a static analysis of the framework $\mathcal{F}$. Our idea is to first to compute a call graph for every entry point of the framework and then to detect whether or not permission checks are present in the call graph. A call graph is a directed graph G containing a set of vertices V representing method calls and a set of arcs A representing links between method calls.

A permission enforcement point (PEP) is a vertex of a call graph whose signature corresponds to a system method which checks permission(s). Each PEP is associated with a list of required permissions $perms_{PEP}$. In Figure 1, the call graph starting from entry point e_4 reaches ck_2, a call to a PEP. To localize in which methods PEP are called, we traverse a call graph $G = (V, A)$ generated from the framework and check whether a vertex V_{PEP} is a PEP. Methods which directly check for permissions are represented as vertice V_{M_i} ($i \in$ {1,2,...,k}), such that $(V_{M_i}, V_{PEP}) \in A$.

We compute one call graph G_i per entry point e_i of the framework ($i \in$ {1,2,...,n})[3]. Then , matrix M is constructed as follows: M is set as a matrix of size (|entry points| × |high level permissions|); all elements of M are initialized to false; for each e_i that reaches one ore more PEP, and for each permission j in $perms_{PEP}$, $M(i, j) = true$. In other terms, M is a condensed version of the reachability information that is latent in call graphs. For instance, a framework with four

[3]This is especially important for field-sensitive static analyzes)

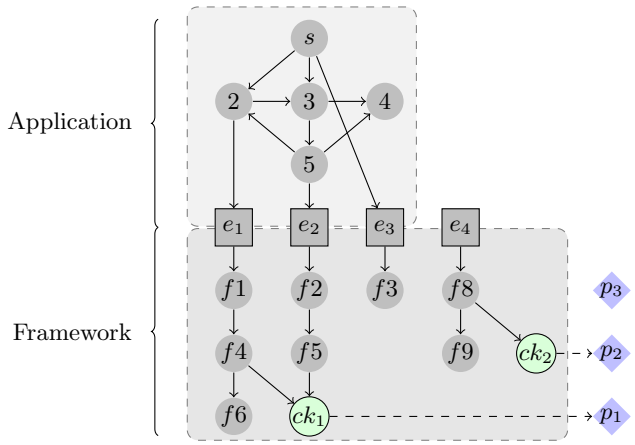

Figure 1: Application and Framework Example

entry points (e_1, e_2, e_3, e_4), and three permissions (p_1, p_2, p_3) is presented in the lower part of Figure 1. For every of those entry points a call graph is constructed. Three of those call graphs have a PEP node: e_1 and e_2 have PEP ck_1 which checks permission p_1 and e_4 has PEP ck_2 which checks permission p_2. On the figure a dashed arrow connects each PEP to the permission(s) it checks. The framework matrix, noted M_{ex}, is then the same as matrix M (see Section 3.2).

Extracting AV simply means listing the list of entry points of a framework $\mathcal{F}$ called by an application *app*. The application example in Figure 1 features a single entry point, s. From s a call graph G_{ex} is generated. All elements of vector AV_{ex} of length $n = 4$ are initially set to $false$: $AV_{ex} = (0, 0, 0, 0)$. Then for every vertex of G_{ex} which is a call to the example framework, the corresponding element of AV_{ex} is set to *true*. In the example, there are three such vertices (represented as entry points e_1, e_2 and e_3 in Figure 1). This leads to the following vector $AV_{ex} = (1, 1, 1, 0)$.

3.4 Computing the Permission Gap

We compute the inferred permission vector according to the definition presented in Section 3.1. The inferred permission list corresponds to permissions set to *true* in IP_{app}. In Figure 1, using matrix M_{ex} and vector AV_{ex} generated above for the example framework and application, we obtain an list of inferred permissions only containing p_1. The permission gap is the difference between the permissions extracted from IP_{app} and the declared permissions $P_d(app)$. If the application declares p_1 and p_2, the permission gap is $\{p_2\}$.

4. OVERVIEW OF ANDROID

This section gives an overview of the architecture and specificity of the Android software stack. We detail how applications access the framework$_{\mathcal{F}}$ and where access control is enforced with respect to permissions.

4.1 Software Stack

Android is a system with different layers. It consists of a modified Linux kernel, C/C++ libraries, a virtual machine called Dalvik, a Java framework and a set of basic applications (including a phone application). Applications for

Android are written in Java. An Android application is packaged into a Android package file (ending in .apk) which contains the Dalvik bytecode, data (pictures, sounds, ...) and the Android manifest file. The developer defines permissions the application may use in this manifest.

An Android application is made of *components* which can be: 1. an *Activity* which is a user interface; 2. a *Service* which runs in background; 3. a *BroadcastReceiver* which listens for "intents" (a kind of message comparable to inter processes communication, aka IPC); 4. a *ContentProvider* which is a kind of backend database used to store and share data[4].

4.2 Services

Applications define services that can be used by other applications. They also communicate with the operating system using a special kind of services called *system services* that are used by the system for enforcing permission checks. System services are specific services running in a specific scope (called the "system server") and allow applications to access system resources. Those resources may be protected by Android permissions. The permission checks associated to services are mostly implemented in Java, but Android also checks permissions in C++ services, content providers and when using intents. In this paper, we focus on the former (Java services), the impact of this focus is discussed in Section 6.

Applications synchronously communicate with others services (deployed from other applications or the OS) through a mechanism called *Binder*. The first step to communicate with a remote service is to dynamically get a reference to the service by calling `Context.getSystemService()` (step 1 in Figure 2). The next step is to call a method on the reference (step 2 in Figure 2). A special component, called "binder" is responsible for delivering references, intercepting and redirecting that service calls to the remote service which performs the actual computation (steps 3 and 5 in Figure 2). The system service is responsible for enforcing the permission policy (step 4 in Figure 2).

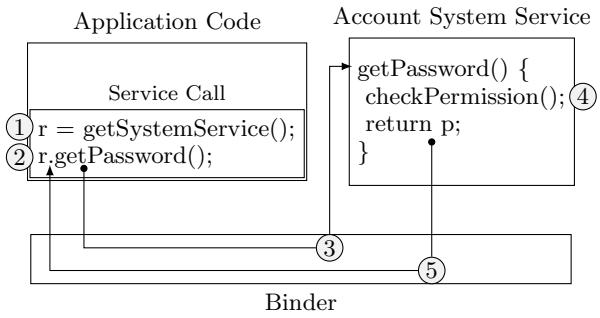

Figure 2: Android System Service

4.3 Permissions and Application Installation

When installing an Android application from an application market, the user has to approve as a whole (or reject as a whole) all the permissions the application declared in its manifest. If all permissions are approved, the application is installed and mapped with the corresponding permissions. Moreover, it receives a device-specific user id (UID) and group memberships for the permissions that are mapped with Unix groups. For instance, an application Foo is given two group memberships `net_bt` and `inet` when associated with permissions `BLUETOOTH` and `INTERNET`, respectively. In other terms, the standard Unix ACL is used as an implementation means for checking permissions.

Android 2.2 declares 107 (115-8) high-level permissions, high-level in the sense that they are enforced at the framework level (ex: to read contact information an application must have the `READ_CONTACT` permission). Note that high-level permissions are related to *where* permissions are checked (in the framework) not *how* (mostly using Unix group memberships by the Dalvik virtual machine). There are eight high-level permissions that may also be indirectly enforced at the kernel level by checking unix group IDs (ex: to create a socket an application has to have the `INTERNET` permission to be in the `inet` group).

In Section 3, we have defined a generic model and methodology to generate a matrix M which maps entry points of a framework$_{\mathcal{F}}$ to permissions. We have seen in Section 4 that the Android system fits in the model and contains a framework corresponding to $_{\mathcal{F}}$. The next Section presents a static analysis to extract M from the Android framework$_{\mathcal{F}}$ and to infer the list of required permissions (as opposed to declared permissions) for an Android application.

5. STATIC ANALYSIS FOR ANDROID

Our approach to detecting permission gaps presented in Section 3 is implemented with two tools. One extracts from a permission-based framework a binary matrix that maps framework methods to permissions, we call it the *mapper*. The other extracts from application code the list of framework methods used, we call it the *sniffer*. In COPES, both tools are based on static analysis. Implementing both tools was much more difficult than expected. In other terms, there was a significant gap between the regularity and the conciseness of the approach presented in Section 3 and the actual implementation.

The key insights of our analysis are related to correctly handling the service and binder mechanisms of Android (see 4.2). This section presents our solutions to the most important issues in order to 1) enable other researchers to replicate our results, and 2) facilitate the implementation of the approach for another permission-based platform.

5.1 Framework Call Graphs

The core of our approach consists of building and manipulating call graphs. COPES call graph construction leverages the Soot call graph analysis Spark [19] together with the service mapping information described in below in 5.3 and 5.4.

We run Spark in context-insensitive, path-insensitive, flow-insensitive, field-sensitive mode to generate the call graph. In context-insensitive mode, every call to a same method are merged to a single edge independently of the context

[4]An application uses URIs (Uniform Resource Identifiers from RFC #2396) to locate and work with local or remote content providers

(receiver and parameters values). A path-insensitive analysis ignores conditional branching hence takes into account all paths of method bodies. The call graph construction is flow-insensitive since it does not consider the order of executions of instructions. It is also field-sensitive because it uses and propagates initialization data (e.g. constructor calls) to reduce the number of egdes.

Spark requires an entry point (usually a main) in order to apply its aggressive edge removal techniques. In the case of an API (such as the Android API), there is no "main". Hence, we build one call graph per public method of the Android API by creating one fake main method per public class of the framework (for Android, `android.*` and `com.android.*`). We can also build an artificial main calling all public methods, which is conceptually equivalent yet less scalable[5].

5.2 Extracting Permission Enforcement Points

Permission Enforcement Points in Android are method calls to certain method of classes `Context` and `ContextWrapper` (for instance method `checkPermission`). Those method calls can be resolved statically. However, the actual permission(s) that are checked are dynamically set by a String parameter or sometimes, an array of strings. Thus, when a check permission system method is found in the call graph, a basic analysis is only able to tell that a permission check occurs, but not which precise permission.

To overcome this issue, we have implemented a String analysis as a Soot plugin. Once PEPs are found, it extracts the corresponding permission(s). This plugin performs an intra-method analysis and manages the following scenarios: either (1) the permission is directly given as parameter, or (2) the permission value is initialized in a variable which is given as a parameter, or (3) an array is initialized with several permissions and is given as a parameter. In every case we do a backward analysis of the method's bytecode using Soot's Unit Graphs which describe relations among statements of a method. In the case where only one permission is given to the method, the first statement in the unit graph containing a reference to a valid Android permission String is extracted and the permission added to the list of the permissions needed by the method under analysis. In case of an array, all permissions of references to Android permission Strings are added to the list.

When no valid permission String is found, methods in the call-stack of the PEP method are analyzed. Indeed, permission String can be assigned indirectly to PEP methods.

5.3 Handling Binder-based Communication

Static analysis can not resolve call to services since they are done dynamically through the binder (see 4). Since the binding uses a lookup table that is instantiated once at boot time within the *system server*, our solution is to intercepted this lookup table and use it in a Soot plugin to redirect every proxy call to the concrete instance of the class which implements the service. In other terms, we feed the call graph engine with this domain specific information that it does not know from code.

[5]we were not able to extract such a call graph on a machine with 24GB RAM

Note that when using a field-sensitive (such as Spark) or context-sensitive analysis, services must be properly initialized. Otherwise, their fields would point to null and method calls on those fields would not be considered during the call graph construction. We resolve this issue by providing a special initialization class to Spark containing services objects towards which remote service calls are redirected.

5.4 Service Identity Inversion

In Android, services can call other services either with the identity of the initial caller (by default) or with the identity of the service itself. In the later case, remote calls are within `clearIdentity()` and `restoreIdentity()` method calls. When using the service identity, the permission checks are not done against the caller's declared permissions, but against the service's declared permissions. Since our goal is to compute the permission gap of an application (and not of system services), we can safely discard all permission checks that occur between calls to `clearIdentity()` and `restoreIdentity()`. This significantly decreases the number of inferred permissions hence the number of false positives.

For instance, let us assume that service S requires and declares permission θ which is not declared by application A. If A calls S, the code of S is executed with the identity of A itself which would require A to declare θ. To avoid this, the portion of code requiring θ is executed with S identity. Spark is flow-insensitive, so when we encounter calls to `clearIdentity()` or `restoreIdentity()`, we use an intra-procedural flow-sensitive analysis to discard permission checks that occur between those calls.

5.5 Reflection in the Framework

If the framework uses reflection, then the call graph construction is incomplete by construction. Fortunately, the Android framework uses reflection in only 7 classes. We manually analyzed their source code. Five of those classes are debugging classes. The *View* class uses reflection for handling animations. Finally, the *VCardComposer* uses reflection in a branch that is only executed for testing purpose. In all cases, the code is not related to system resources hence no permission checks are done at all. This does not impact the static analysis of the Android framework.

5.6 Dynamic Class Loading

The Java language has the possibility to load classes dynamically. When used this features makes static analysis impossible since the loaded classes are only known at runtime. We found that eight classes of the Android system are using the `loadClass` method. After manual check, six of them are system management classes and either are not linked to permission checks (ex: instrumenting an application) or have to be accessed through a service. Two are related to the *webkit* packaged. They are used in the `LoadFile` and `PluginManager` classes. In both cases, permissions are checked *before* loading classes, and not inside the loaded classes. Thus, there is no missed permission enforcement points either.

5.7 Bytecode Manipulation Toolkits

A last technical yet blocking issue was related to manipulation of Android bytecode. We had to write the mapper and

	SOOT Spark with binder
# methods	126660
# permissions	71
# methods with no check	112824
# meth. with $\geq$ 1 perm.	9562
median perm. checks	2
max perm. checks	50
# perm. checks	137408

Table 1: Descriptive Statistics of The Permission Maps Found by Static Analysis

the sniffer on top of two different toolkits: the mapper uses the Soot analysis framework developed at McGill University [29]; the sniffer uses the ASM framework [2]. We had to use two different toolkits for the following reasons. On the one hand, the code of the framework is open-source and written in Java, which is perfectly appropriate for an analysis using Soot. On the other hand, since we do not assume to have the source code of end-user commercial applications, the application bytecode is only available as Dalvik bytecode. While we can transform Dalvik bytecode to Java bytecode using a tool called "ded" developed at Penn State University[6], the bytecode resulting from too many complex transformations is often not compatible with Soot for obscure reasons. The lower-level API of ASM enabled us to overcome these problems.

5.8 Recapitulation

We have presented the core technical issues we encountered while implementing our approach. We think that those problems may arise in other permission-based platforms than Android, and that identifying them and their solutions can be of great help for future work. Last not but not least, those points are crucial for replication of our results.

6. EVALUATION

This section presents an evaluation of our approach. First, we discuss the permission map extracted by static analysis using Soot and Spark. Then we compare our results to the map extracted by Felt et al.[10] using runtime testing techniques. Finally, we show that our approach actually detects permission gaps in real applications published in two different application stores.

6.1 Extracted Permission Maps

In the Android v2.2 framework, 115 permissions are defined. When predicting the required permissions of Android applications, we want to guarantee that the inferred permission set is sound, i.e. that all inferred permissions are actually checked in code and that we do not miss some checks. As said in Section 3.1 (definition: Low Level Permission), and in Section 4.2, our static analysis method does not deal with: 8 low-level kernel permissions; 30 permissions checked at the level C++ services; 8 permissions checked at the level of content provider. Removing these permissions from the initial set of 115 permissions and by taking care of overlapping permissions (for instance, a permission can be checked at both C++ service and content provider levels) yields a set of *71 high-level permissions*. In the following, our discussion and comparison will only consider this set of 71 permissions.

For those 71 high level permissions, we claim that our static analysis at the framework level is sound.

We ran the static analysis based on Soot and which uses the Spark call graph analysis described in 5.1 augmented with binder and service specifities (see Sections 5.3 and 5.4) on the Android v2.2 framework bytecode. The resulting map is summarized in table 1. It gives the number of analyzed entry points (methods of the framework), the number of methods with no permission checks, the total number of permission checks (ones in the matrix), and the number of methods with at least one permission checks (with the median and maximum number of permission checks).

According to this analysis, there are 9562 methods requiring at least one permission, and among them, there are a median of 2 permissions checked.

This fits with our developer experience with Android, the methods have a clear scope and generally require a few permissions (for instance, a method related to Bluetooth management only requires permission BLUETOOTH). The maximum of 50 permissions is related to methods which are highly dependent of the usage context. In practice, developers only use the method indirectly and in a specific context and declare a handful permission. However, from the blind viewpoint of static analysis, there are 50 permissions involved in this method. There are only couple of such outlier in the 9562 methods predicted by Spark to require at least one permission. The question whether we are still sound, i.e. whether we did not remove too many edges in the call graph is answered in the next sub-sections.

The matrice is very sparse (it mostly contains zeros and a few ones – the number of permission checks), because many methods do not contain permission checks and because one method checks at most an handful permissions.

In terms of CPU cost, the computation of the most CPU-intensive analysis, Soot Spark, is performed in about 11 hours on a Desktop Dell dual quad-core 2.4GHz with 24 Go RAM.

6.2 Comparison with Felt et al.

Let us now compare our results obtained with static analysis with the results of Felt et al.' obtained with testing [10]. Both extract a list of required permissions for each method of the Android framework. Felt et al.'s results contain 673 methods related with high-level permissions. We analyze only 671 methods because 2 methods are related with application-specific objects provided in Felt's approach that are not available per construction in our static analysis approach. Using our Spark-based static analysis approach with a maximum call graph depth of 10, for a given method, we either find the same permission set, or a larger one. Our method never misses a permission that Felt et al. describe, this is piece of evidence of the soundness of our approach.

More precisely, we infer the same permission set per method signature for 552 methods (82.3% of commonly analyzed

[6] `http://siis.cse.psu.edu/ded/`

Permission set	Number of Methods
#Methods analyzed in [10]	1282
#Methods with HL perm. only	673
Identical	552 (82.3%)
we find more permission checks	119 (17.7%)
one more	118 (17.6%)
two more	1 (0.1%)
[soundness evidence] we find less permission checks	0 (0%)

Table 2: Comparison with Felt et al. [10]. The discrepancy is due to the conceptual differences between static analysis and testing.

methods). There is one ore additional permissions for 119 methods (1 additional permission for 118 methods, 2 for 1 methods).There is no method for which we miss a permission, Table 2 summarizes those results. Let us now discuss the discrepancy between our results.

The additional permissions are due to either analyzing irrelevant code or to missing input data in Felt et al.'s approach. In the latter case, we are able to find permissions that are checked within specific contexts that were not taken into account by the generated tests of Felt et al. For instance, `MOUNT_UNMOUNT_FILESYSTEMS` is only checked for method `MountService.shutdown()` if the media (storage device) is "*present not mounted and shared via USB mass storage*" (from the API documentation). Another permission, `READ_PHONE_STATE` is needed for method `CallerInfo.getCallerId()` only if the phone number passed in parameter is the voice mail number. Those test cases were not generated by Felt's testing approach. In real applications, test generation techniques can not guarantee a comprehensive exploration of the input space.

To us, these findings are typical when comparing a static analysis approach against a testing one: static analysis sometimes suffers from analyzing all code (including debugging and dead code, or code run in specific runtime environments), but is strong at abstracting over input data. On the other hand, testing must simulate as close as possible the production environment, but is cursed to always miss very specific usage scenarios.

Those results highlight the complementarity between static analysis and testing in the context of permission inference. We think that the static analysis approach is complementary to the testing approach. Indeed, the testing approach yields an under-approximation which misses some permission checks whereas the static analysis approach yields an over-approximation in which those missing permission checks are found. Using both approaches in collaboration would enable developers to obtain a lower and a upper bound of the permission gap. In particular, for an given Android applications, if both testing and static analysis approaches yield the same list of permissions, this list is the exact list of required permissions. This strong result is only possible by using both approaches in conjunction.

Note that we also compared the Spark based static analysis with a naive (Class Hierachy Analysis based) one which yields worst results (bigger permission sets). As expected, these results show that the precision is higher when using Spark.

6.3 Permission Gaps in Real Applications

We ran our tool on two datasets of Android applications. The first comes from an alternative Android Market[7] and contains 1329 android applications. For the second one, we consider the top 50 download applications of all 34 top-level categories of the Official Android Market, as well as the top 500 of all the applications and the top 500 of new applications (at the date of February, 23^{rd} 2012). As a result, after deduplicating the applications that appear in several rankings, the second dataset contains 2057 applications.

Alternative Android Market: For sake of soundness, we discard 587 applications that use reflection and/or class loading. Of the 742 remaining applications, 94 are declaring one or more permissions which they do not use. Consequently, we identify a permission gap for 94 Android applications. We define the "area of the attack surface" with respect to permission gaps, as the number of unnecessary permission. In all, among applications suffering from a permission gap, 76.6% have an attack surface of 1 permission, 19.2% have an attack surface of 2 permissions, 2,1% of 3 permissions and also 2,1% of 4 permissions.

Official Android Market: For sake of soundness, we discard 1378 applications using reflection and/or class loading. On the 679 remaining applications, 124 are declaring one or more permissions which they do not use. In all, among applications suffering from a permission gap, 64.5% have an attack surface of 1 permission, 23.4% have an attack surface of 2 permissions, 12.1% of 3 or more permissions.

To sum up, those results show that permission gaps exists, and that our tool allows developers to fix the declared permission list in order to reduce the attack surface of permission-based software.

7. RELATED WORK

We have presented an approach to reduce the attack surface of permission-based software. The concept of "attack surface" was introduced by Manadhata and colleagues [20], it describes all manners *in which an adversary can enter the system and potentially cause damage*. This paper describes a method to identify the attack surface of Android applications, which is a important research challenge given the sheer popularity of the Android platform. In the context of Android, reducing the attack surface is adhering to the principle of least privileges introduced by Saltzer [27].

7.1 On the Java Permission Model

While the Android permission model is different from the one implemented in Java, the following pieces of research present related and relevant points to put our contribution in perspective.

Koved and al. described a new static analysis [18] to generate a permission list for a Java2 program (in the Java permission model). An improved methodology was presented by Geay et al. [14]. We also use static analysis but in

[7]www.freewarelovers.com/android

the context of Android which differs from a Java environment especially with respect to the binder mechanism linking Android API to services. As shown in our evaluation, the binder prevents off-the-shelf Java static analysis tools to resolve remote call to a service.

Pistoia et al. [25] presented a static analysis to identify portions of the code which should be made privileged. This issue does not arise in the Android framework since code is not privileged per se, the access control is instead done at entry points. This means that the Android framework designers must be careful of creating unique entry points protected by permission enforcement points, but does not impact our static analysis.

Role-based access control (RBAC) mechanisms are analyzed using static analysis by Centonze et al. [4]. When a protected operation manipulates data, this data should not be directly or indirectly accessible by a path not defined in the policy. If not, the operation is said to be *location-inconsistent*. The tool they developed can check whether or not an RBAC policy for JavaEE programs is location consistent or present some flaws. The Android system defines permissions which protects operation which in turn manipulate protected data. Our goal consists of computing permission gaps which may reveal a violation of the principle of least privilege. Whether Android protected operations are location consistent is out of scope of this paper.

Also related to role-based access control, Pistoia et al. [24] formally model RBAC and statically check the consistency of a JavaEE based RBAC system. We check that permission lists of Android applications respect the principle of least priviledge. The concepts are the same (Android permissions could be approximated to roles, and we check which roles are needed at every point of the Android framework) but the target systems are not. Interestingly, we use a similar approach for solving the Binder problem as they do for solving the remote method invocation problem: instead of statically analyzing the Binder/RMI code which would not resolve the method, a mapping is computed from the call to a remote method to the remote method itself. A major difference though is that in the case of Android system services and context must be initialized beforehand to simulate a correct system state.

7.2 On the Android Permission Model

The Android security model has been described as much in the gray literature [9, 28] as in the official documentation [16]. Different kinds of issues have been studied such as social engineering attacks [17], collusion attacks [21], privacy leaks [15] and privilege escalation attacks [12, 6]. In contrast, this paper does not describe a particular weakness but rather a software engineering approach to reduce potential vulnerabilities.

However, we are not describing a new security model for Android as done by [22, 23, 7, 5, 3]. For instance, Quire [7] maintains at runtime the call chain and data provenance of requests to prevent certain kinds of attacks. In this paper, we do not modify the existing Android security model and we devise an approach to mitigate its intrinsic problems.

Also, different authors empirically explored the usage of the Android model. For instance, Barrera et al. [1] presented an empirical study on how permissions are used. In particular, they used visualizing techniques such as self-organizing maps to identify patterns of permissions depending on the application domain, and patterns of permission grouping. Other empirical studies include Felt's one [11] on the effectiveness of the permission model, and Roesner's one [26] on how users react to permission-based systems. While our paper also contains an empirical part, it is also operational because we devise an operational software engineering approach to tame permission-based security models in general and Android's one in particular.

Enck et al [8] presented an approach to detect dangerous permissions and malicious permission groups. They devised a language to express rules which are expressed by security experts. Rules that do not hold at installation time indicate a potential security problem hence a high attack surface. Our goal is different, we don't aim at identifying risks identified from experts, but to identify the gap between the application's permission specification and the actual usage of platform resources and services. Contrary to [8], our approach is fully automated and does not involve an expert in the process.

Finally, Felt et al. [10] concurrently worked on the same topic as this paper. They published a very first version of the map between developer's resources (e.g. API calls) and permissions. Interestingly, we took two completely different approaches to identify the map: while they use testing, we use static analysis. As a result, our work validates most of their results although we found several discrepancies that we discussed in details in Section 6. But the key difference is that our approach is fully automated while theirs requires manually providing testing "seeds" (such as input values). However, in the presence of reflection, their approach works better if the tests are appropriate. Hence, we consider that both approaches are complementary, both at the conceptual level for permission-based architectures, and concretely for reverse-engineering and documenting Android permissions.

8. CONCLUSIONS AND PERSPECTIVES

In this paper, we have presented a generic approach to reduce the attack surface of permission-based software. We have extensively discussed the problematic consequences of having more permissions than necessary and showed that the problem can be mitigated using static analysis. The approach has been fully implemented for Android, a permission-based platform for mobile devices. Our prototype implementation is able to automatically find 9562 Android framework entry points which check permissions. In a permission-based framework, all those checks have to be documented, hence our approach does a significant job in achieving this task in a systematic manner. For end-user applications, our evaluation revealed that 94/742 and 35/679 crawled applications from application stores for Android indeed suffer from permission gaps. We have also shown that our static analysis based approach is complementary to concurrent work [10] based on testing.

The security architecture of permission based software in general and Android in particular is complex. In this paper, we abstracted over several characteristics of the platform such as low-level permissions. We are now working on a modular approach that would be able to analyze native code and bytecode in concert and to combine the permission information from both. Furthermore, we are exploring how to express permission enforcement as a cross cutting concern, in order to automatically add or remove permis-

sion enforcement points at the level of application or the framework, according to a security specification.

9. REFERENCES

[1] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *ACM Conference on Computer and Communications Security (CCS 2010)*, pages 73–84, Chicago, Illinois, USA, October 4-8, 2010.

[2] E. Bruneton. Asm 3.0, a java bytecode engineering library, http://download.forge.objectweb.org/asm/asm-guide.pdf, 2007.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, Apr 2011.

[4] P. Centonze, G. Naumovich, S. J. Fink, and M. Pistoia. Role-based access control consistency validation. In *ISSTA 2006*, pages 121–132.

[5] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th International Conference on Information security*, 2011.

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, 2011.

[7] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, Aug. 2011.

[8] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM CCS*, pages 235–245, New York, NY, USA, 2009.

[9] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security and Privacy*, 2009.

[10] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS 2011*.

[11] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[12] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[13] Gartner.com. Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent. `http://goo.gl/HkyA4`, Last accessed: March 2 2012.

[14] E. Geay, M. Pistoia, T. Tateishi, B. G. Ryder, and J. Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *ICSE*, pages 177–187. IEEE, 2009.

[15] C. Gibler, J. Crussel, J. Erickson, and H. Chen. Androidleaks detecting privacy leaks in android applications. Technical report, UC Davis, 2011.

[16] Google. The android developer's guide, last-accessed: 2011-09. `http://developer.android.com/guide/index.html`.

[17] S. Hoffman. Zeus banking trojan variant attacks android smartphones. *CRN*, 2011. `http://goo.gl/xAEGr`.

[18] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. *ACM SIGPLAN Notices*, 37(11):359–372, Nov. 2002.

[19] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *12th International Conference on Compiler Construction*, 2003.

[20] P. Manadhata and J. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371 –386, may-june 2011.

[21] C. Marforio, A. Francillon, and S. Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, April 2011.

[22] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[23] M. Ongtang, S. McLaughlin, W. Enck, and P. McDanie. Semantically rich application-centric security in android. *Journal of Security and Communication Networks*, 2011.

[24] M. Pistoia, S. J. Fink, R. J. Flynn, and E. Yahav. When role models have flaws: Static validation of enterprise security policies. In *ICSE*, 2007.

[25] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, 2005.

[26] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. Technical Report MSR-TR-2011-91, Microsoft Research, 2011.

[27] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, 1975.

[28] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. Google android: A state-of-the-art review of security mechanisms. *CoRR*, abs/0912.5101, 2009.

[29] R. Vallée-Rai, L. Hendren, V. Sundaresan, E. G. Patrick Lam, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.